PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem

Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu,
Vijay Subramanian, Fred Baker and Bill VerSteeg
Advanced Architecture & Research Group, Cisco Systems Inc., San Jose, CA 95134, U.S.A.
{ropan, prenatar, cpiglion, mysuryan, vijaynsu, fred, versteb}@cisco.com

Abstract—Bufferbloat is a phenomenon where excess buffers in the network cause high latency and jitter. As more and more interactive applications (e.g. voice over IP, real time video conferencing and financial transactions) run in the Internet, high latency and jitter degrade application performance. There is a pressing need to design intelligent queue management schemes that can control latency and jitter; and hence provide desirable quality of service to users.

We present here a lightweight design, PIE (Proportional Integral controller Enhanced), that can effectively control the average queueing latency to a reference value. The design does not require per-packet extra processing, so it incurs very small overhead and is simple to implement in both hardware and software. In addition, the design parameters are self-tuning, and hence PIE is robust and optimized for various network scenarios. Simulation results, theoretical analysis and Linux testbed results show that PIE can ensure low latency and achieve high link utilization under various congestion situations.

Index Terms—bufferbloat, Active Queue Management (AQM), Quality of Service (QoS), Explicit Congestion Notification (ECN)

I. INTRODUCTION

The explosion of smart phones, tablets and video traffic in the Internet brings about a unique set of challenges for congestion control. To avoid packet drops, many service providers or data center operators require vendors to put in as much buffer as possible. With rapid decrease in memory chip prices, these requests are easily accommodated to keep customers happy. However, the above solution of large buffers fails to take into account the nature of TCP, the dominant transport protocol running in the Internet. The TCP protocol continuously increases its sending rate and causes network buffers to fill up. TCP cuts its rate only when it receives a packet drop or mark that is interpreted as a congestion signal. However, drops and marks usually occur when network buffers are full or almost full. As a result, excess buffers, initially designed to avoid packet drops, would lead to highly elevated queueing latency and jitter. The phenomenon was detailed in 2009 [1] and the term, "bufferbloat" was introduced by Jim Gettys in late 2010 [2].

Figure 1 shows an example of extremely long latencies that are caused by the bufferbloat problem. Ping messages were sent overnight from a hotel in Ireland to San Jose, CA on January 27, 2012. Figure 1 depicts frequent delays in the neighborhood of 8 to 9 seconds. A review of the round trip time distribution, shown in Figure 2, is instructive. It suggests that as many as eight copies of the same TCP segment, spaced Retransmission Time Out (RTO) apart, were present at some time in the hotel's DSL service. This obviously reduces effective bandwidth; any bandwidth used for an unnecessary retransmission is unavailable for valuable data. It also reduces the value of the service and the willingness of a consumer to pay for it. If we want to see additional value-add services, such as high volume video content delivery in these networks, we need to get control of the data buffered in networks.

Active queue management (AQM) schemes, such as RED [3], BLUE [4], PI [5], AVQ [6], etc, have been around for well over a decade. AQM schemes could potentially solve the aforementioned
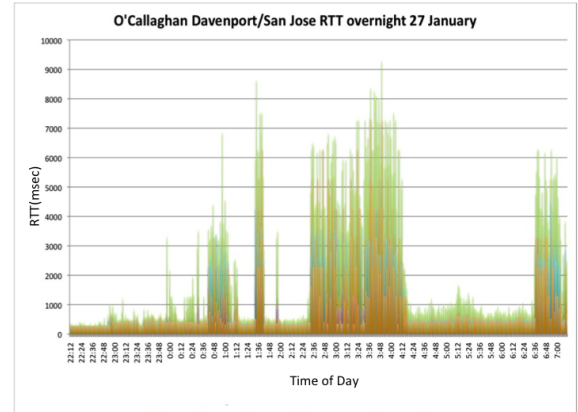

Fig. 1. An Example of Extreme Long Latency: RTTs measured using ping messages sent overnight from a hotel in Ireland to San Jose, CA.
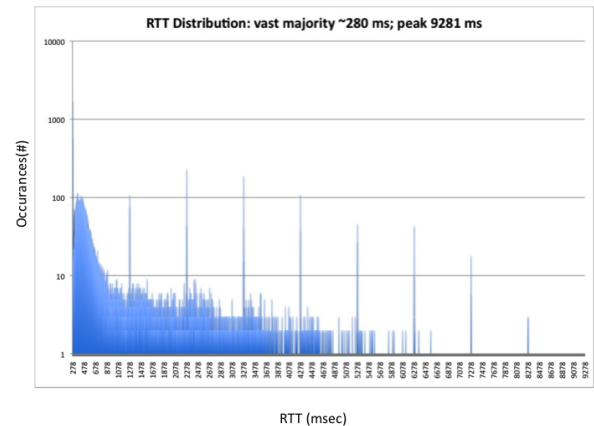

Fig. 2. Experiment RTT Distributions: number of occurrences as a function of RTTs. There are spikes that are RTOs apart.

problem. RFC 2309 [7] strongly recommends the adoption of AQM schemes in the network to improve the performance of the Internet. RED is implemented in a wide variety of network devices, both in hardware and software. Unfortunately, due to the fact that RED needs careful tuning of its parameters for various network conditions, most network operators do not turn RED on. In addition, RED is designed to control the queue length which would affect delay implicitly. It does not control latency directly.

Note that the delay bloat caused by poorly managed big buffer is really the issue here. If latency can be controlled, bufferbloat, i.e., adding more buffers for bursts, is not a problem. More buffer space would allow larger bursts of packets to pass through as long

as we control the average queueing delay to be small. Unfortunately, Internet today still lacks an effective design that can control buffer latency to improve the quality of experience to latency-sensitive applications. In addition, it is a delicate balancing act to design a queue management scheme that not only allows short-term burst to smoothly pass, but also controls the average latency when long-term congestion persists.

Recently, a new AQM scheme, CoDel [8], was proposed to control the latency directly to address the bufferbloat problem. CoDel requires per packet timestamps. Also, packets are dropped at the dequeue function after they have been enqueued for a while. Both of these requirements consume excessive processing and infrastructure resources. This consumption will make CoDel expensive to implement and operate, especially in hardware.

In this paper, we present a lightweight algorithm, PIE (Proportional Integral controller Enhanced), which combines the benefits of both RED and CoDel: easy to implement like RED while directly control latency like CoDel. Similar to RED, PIE randomly drops a packet at the onset of the congestion. The congestion detection, however, is based on the queueing latency like CoDel instead of the queue length like conventional AQM schemes such as RED. Furthermore, PIE also uses the latency moving trends: latency increasing or decreasing, to help determine congestion levels.

Our simulation and lab test results show that PIE can control latency around the reference under various congestion conditions. It can quickly and automatically respond to network congestion changes in an agile manner. Our theoretical analysis guarantees that the PIE design is stable for arbitrary number of flows with heterogeneous round trip times under a predetermined limit.

In what follows, Section II specifies our goals of designing the latency-based AQM scheme. Section III explains the scheme in detail. Section IV presents simulation and lab studies of the proposed scheme. In Section V, we present a control theory analysis of PIE. Section VI concludes the paper and discusses future work.

II. DESIGN GOALS

We explore a queue management framework where we aim to improve the performance of interactive and delay-sensitive applications. The design of our scheme follows a few basic criteria.

- *Low Latency Control.* We directly control queueing latency instead of controlling queue length. Queue sizes change with queue draining rates and various flows' round trip times. Delay bloat is the real issue that we need to address as it impairs real time applications. If latency can be controlled to be small, bufferbloat is not an issue. As a matter of fact, we would allow more buffers for sporadic bursts as long as the latency is under control.

- *High Link Utilization.* We aim to achieve high link utilization. The goal of low latency shall be achieved without suffering link under-utilization or losing network efficiency. An early congestion signal could cause TCP to back off and avoid queue buildup. On the other hand, however, TCP's rate reduction could result in link under-utilization. There is a delicate balance between achieving high link utilization and low latency.

- *Simple Implementation.* The scheme should be simple to implement and easily scalable in both hardware and software. The wide adoption of RED over a variety of network devices is a testament to the power of simple random early dropping/marking. We strive to maintain similar design
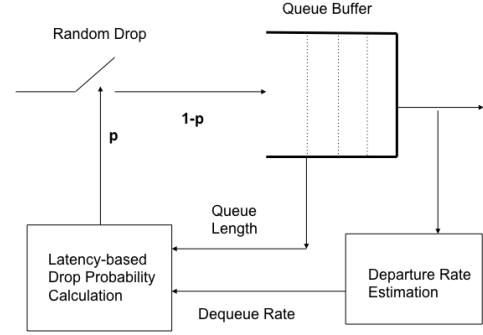

Fig. 3. Overview of the PIE Design. The scheme comprises three simple components: a) random dropping at enqueuing; b) latency based drop probability update; c) dequeuing rate estimation.

simplicity.

- *Guaranteed stability and Fast Responsiveness.* The scheme should ensure system stability for various network topologies and scale well with arbitrary number streams. The system also should be agile to sudden changes in network conditions. Design parameters shall be set automatically. One only needs to set performance-related parameters such as target queue delay, not design parameters.

We aim to find an algorithm that achieves the above goals. It is noted that, although important, fairness is orthogonal to the AQM design whose primary goal is to control latency for a given queue. Techniques such as Fair Queueing [9] or its approximate such as AFD (Approximate Fair Dropping) [10] can be combined with any AQM scheme to achieve fairness. Therefore, in this paper, we focus on controlling a queue's latency and ensuring flows' fairness is not worse than those under the standard DropTail or RED design.

III. THE PIE SCHEME

In the section, we describe in detail the design of PIE and its operations. As illustrated in Figure 3, our scheme comprises three simple components: a) random dropping at enqueuing; b) periodic drop probability update; c) dequeuing rate estimation.

The following subsections describe these components in further detail, and explain how they interact with each other. At the end of this section, we will discuss how the scheme can be easily augmented to precisely control bursts.

A. Random Dropping

Like most state-of-the-art AQM schemes, PIE would drop packets randomly according to a drop probability, p, that is obtained from the "drop probability calculation" component. No extra step, like timestamp insertion, is needed. The procedure is as follows:

Random Dropping:
Upon packet arrival
randomly drop a packet with a probability p.

B. Drop Probability Calculation

The PIE algorithm updates the drop probability periodically as follows:

- estimate current queueing delay using Little's law:
  $$cur\_del = \frac{qlen}{avg\_drate};$$

- calculate drop probability p as:
  $$p = p + \alpha * (cur\_del - ref\_del) + \beta * (cur\_del - old\_del);$$

- update previous delay sample as:
  $$old\_del = cur\_del.$$

The average draining rate of the queue, avg_drate, is obtained from the "departure rate estimation" block. Variables, cur_del and old_del, represent the current and previous estimation of the queueing delay. The reference latency value is expressed in ref_del. The update interval is denoted as T_{update}. Parameters α and β are scaling factors.

Note that the calculation of drop probability is based not only on the current estimation of the queueing delay, but also on the direction where the delay is moving, i.e., whether the delay is getting longer or shorter. This direction can simply be measured as the difference between cur_del and old_del. Parameter α determines how the deviation of current latency from the target value affects the drop probability; β exerts the amount of additional adjustments depending on whether the latency is trending up or down. The drop probability would be stabilized when the latency is stable, i.e. cur_del equals old_del; and the value of the latency is equal to ref_del. The relative weight between α and β determines the final balance between latency offset and latency jitter. This is the classic Proportional Integral controller design [11], which has been adopted for controlling the queue length before in [5] and [12]. We adopt it here for controlling queueing latency. In addition, to further enhance the performance, we improve the design by making it auto-tuning as follows:

if $p < 1\%$: $\alpha = \tilde{\alpha}/8$; $\beta = \tilde{\beta}/8$;
else if $p < 10\%$: $\alpha = \tilde{\alpha}/2$; $\beta = \tilde{\beta}/2$;
else: $\alpha = \tilde{\alpha}$; $\beta = \tilde{\beta}$;

where $\tilde{\alpha}$ and $\tilde{\beta}$ are static configured parameters. Auto-tuning would help us not only to maintain stability but also to respond fast to sudden changes. The intuitions are the following: to avoid big swings in adjustments which often leads to instability, we would like to tune p in small increments. Suppose that p is in the range of 1%, then we would want the value of α and β to be small enough, say 0.1%, adjustment in each step. If p is in the higher range, say above 10%, then the situation would warrant a higher single step tuning, for example 1%. The procedures of drop probability calculation can be summarized as follows.

Drop Probability Calculation:

Every T_{update} interval

1. Estimation current queueing delay:
   $$cur\_del = \frac{qlen}{avg\_drate}.$$

2. Based on current drop probability, p, determine suitable step scales:

$$
\begin{aligned}
if\ p\ &<\ 1\%, & \alpha =\ &\tilde{\alpha}/8; & \beta = \tilde{\beta}/8; \\
else\ if\ p\ &<\ 10\%, & \alpha =\ &\tilde{\alpha}/2; & \beta = \tilde{\beta}/2; \\
else\ , & & \alpha =\ &\tilde{\alpha}; & \beta = \tilde{\beta};
\end{aligned}
$$

3. Calculate drop probability as:
   $$p = p + \alpha * (cur\_del - ref\_del) + \beta * (cur\_del - old\_del);$$

4. Update previous delay sample as:
   $$old\_del = cur\_del.$$

We have discussed packet drops so far. The algorithm can be easily applied to networks codes where Early Congestion Notification (ECN) is enabled. The drop probability p could simply mean marking probability.

C. Departure Rate Estimation

The draining rate of a queue in the network often varies either because other queues are sharing the same link, or the link capacity fluctuates. Rate fluctuation is particularly common in wireless networks. Hence, we decide to measure the departure rate directly as follows:

Departure Rate Calculation:

Upon packet departure

1. Decide to be in a measurement cycle if:
   $$qlen > dq\_threshold;$$

2. If the above is true, update departure count dq_count:
   $$dq\_count = dq\_count + dq\_pktsize;$$

3. Update departure rate once $dq\_count > dq\_threshold$ and reset counters:

$$
\begin{aligned}
dq\_int &= now - start; \\
dq\_rate &= \frac{dq\_count}{dq\_int}; \\
avg\_drate &= (1 - \varepsilon) * avg\_drate + \varepsilon * dq\_rate \\
start &= now. \\
dq\_count &= 0;
\end{aligned}
$$

From time to time, short, non-persistent bursts of packets result in empty queues, this would make the measurement less accurate. Hence we only measure the departure rate, dq_rate, when there are sufficient data in the buffer, i.e., when the queue length is over a certain threshold, $dq\_threshold$. Once this threshold is crossed, we obtain a measurement sample. The samples are exponentially averaged, with averaging parameter ε, to obtain the average dequeue rate, avg_drate. The parameter, dq_count, represents the number of bytes departed since the last measurement. The threshold is recommended to be set to 10KB assuming a typical packet size of around 1KB or 1.5KB. This threshold would allow us a long enough period, dq_int, to obtain an average draining rate but also fast enough to reflect sudden changes in the draining rate. Note that this threshold is not crucial for the system's stability.

D. Handling Bursts

The above three components form the basis of the PIE algorithm. Although we aim to control the average latency of a congested queue, the scheme should allow short term bursts to pass through the system without hurting them. We would like to discuss how PIE manages bursts in this section.

Bursts are well tolerated in the basic scheme for the following reasons: first, the drop probability is updated periodically. Any short term burst that occurs within this period could pass through without incurring extra drops as it would not trigger a new drop probability calculation. Secondly, PIE's drop probability calculation is done incrementally. A single update would only lead to a small incremental change in the probability. So if it happens that a burst does occur at the exact instant that the probability is being calculated, the incremental nature of the calculation would ensure its impact is kept small.

Nonetheless, we would like to give users a precise control of the burst. We introduce a parameter, max_burst, that is similar to the burst tolerance in the token bucket design. By default, the parameter is set to be 100ms. Users can certainly modify it according to their application scenarios. The burst allowance is added into the basic PIE design as follows:

Burst Allowance Calculation:

Upon packet arrival

1. If $burst\_allow > 0$

$$enque\ packet\ bypassing\ random\ drop;$$

Upon dq_rate update

2. Update burst allowance:

$$burst\_allow = burst\_allow - dq\_int;$$

3. if $p = 0$; and both cur_del and old_del less than $ref\_del/2$, reset $burst\_allow$,

$$burst\_allow = max\_burst;$$

The burst allowance, noted by $burst\_allow$, is initialized to max_burst. As long as $burst\_allow$ is above zero, an incoming packet will be enqueued bypassing the random drop process. Whenever dq_rate is updated, the value of $burst\_allow$ is decremented by the departure rate update period, dq_int. When the congestion goes away, defined by us as p equals to 0 and both the current and previous samples of estimated delay are less than $ref\_del/2$, we reset $burst\_allow$ to max_burst.

IV. PERFORMANCE EVALUATION

We evaluate the performance of the PIE scheme in both ns-2 simulations and testbed experiment using Linux machines. As the latest design CoDel is in Linux release, we compare PIE's performance against RED in simulations and against CoDel in testbed evaluations.

A. Simulations Evaluation

In this section we present our ns-2 [13] simulations results. We first demonstrate the basic functions of PIE using a few static scenarios; and then we compare PIE and RED performance using dynamic scenarios. We focus our attention on the following performance metrics: instantaneous queue delay, drop probability, and link utilization.

The simulation setup consists of a bottleneck link at 10Mbps with a RTT of 100ms. Unless otherwise stated the buffer size is 200KB. We use both TCP and UDP traffic for our evaluations. All TCP traffic sources are implemented as TCP New Reno with SACK running an FTP application. While UDP traffic is implemented using Constant Bit Rate (CBR) sources. Both UDP and TCP packets are configured to have a fixed size of 1000B. Unless otherwise stated the PIE parameters are configured to

(a) Light 5 TCP Flows (b) Heavy 50 TCP Flows
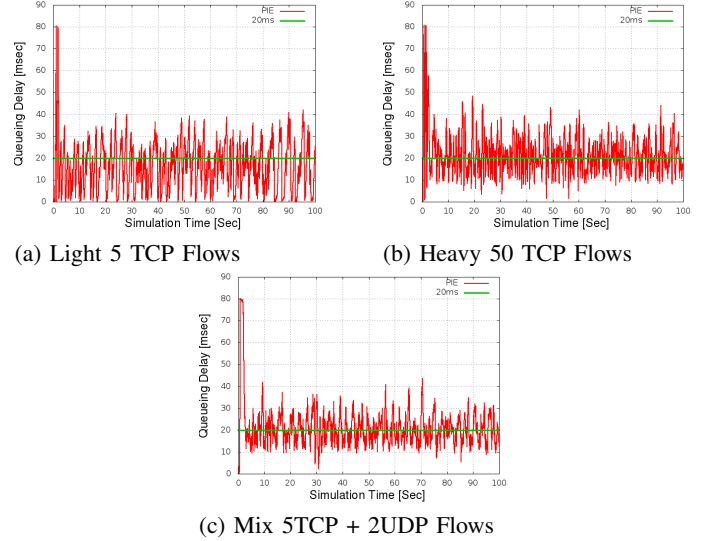

(c) Mix 5TCP + 2UDP Flows

Fig. 4. Queueing Latency Under Various Traffic Loads: a) 5 TCP flows; b) 50 TCP flows; c) 5 TCP + 2 UDP flows. Queueing latency is controlled at the reference level of 20ms regardless of the traffic intensity.

their default values; i.e., $delay\_ref$ = 20ms, T_{update} = 30ms, $\tilde{\alpha}$ = 0.125Hz, $\tilde{\beta}$ = 1.25Hz, $dq\_threshold$ = 10KB, max_burst = 100ms.

(a) Light 5 TCP Flows (b) Heavy 50 TCP Flows
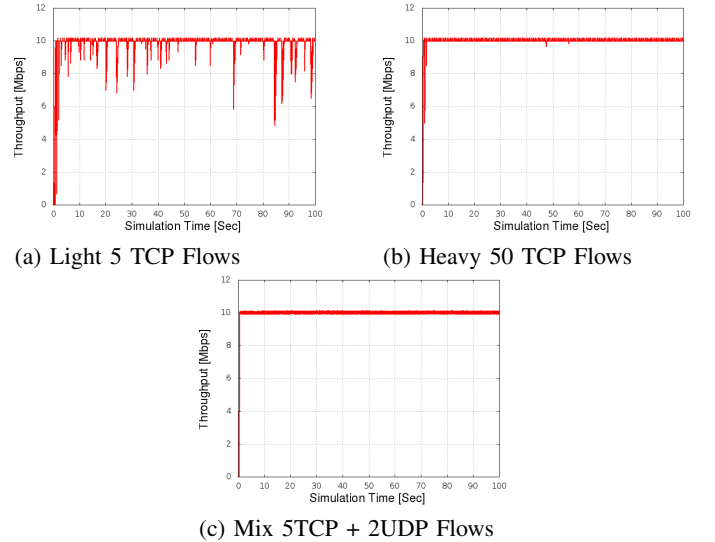

(c) Mix 5TCP + 2UDP Flows

Fig. 5. Link Throughput Under Various Traffic Loads: a) 5 TCP flows; b) 50 TCP flows; c) 5 TCP + 2 UDP flows. High link utilization is achieved regardless of traffic intensity, even under low multiplexing case.

Function Verification: We first validate the functionalities of PIE, making sure it performs as designed using static traffic sources with various loads.

1) Light TCP traffic: Our first simulation evaluates PIE's performance under light traffic load of 5 TCP flows. Figure 4(a) shows the queueing delay, and Figure 5(a) plots the instantaneous throughput respectively. Figure 4(a) demonstrates that the PIE algorithm is able to maintain the queue delay around the equilibrium reference of 20ms. Due to low multiplexing, TCP's sawtooth behavior is still slightly visible in Figure 4(a). Nonetheless, PIE regulates the TCP traffic quite well so that the link is close to its full capacity as shown in Figure 5(a). The average total throughput is 9.82Mbps. Individual
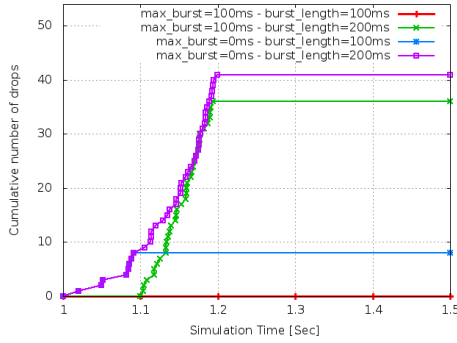
Fig. 6. PIE's Burst Control: at simulation time 1s, the UDP flow starts sending traffic. With a burst allowance of 100ms, the flow can either pass through without incurring drops or starting incurring drops at 1.1sec. With a burst allowance of 0ms, the flow would incur drops right from the beginning. The longer the burst is, the higher the number of drops is.
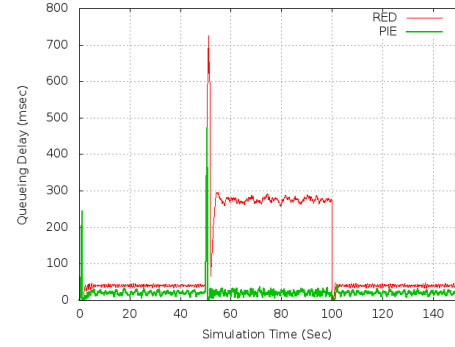

Fig. 7. PIE vs. RED Performance Comparison Under Varying Link Capacity: 0s-50s, link capacity = 100Mbps, 50s-100s, link capacity = 20Mbps, 100s-150s, link capacity = 100Mbps. By only controlling the queue length, RED suffers long queueing latency when the queue draining rate changes. PIE is able to quickly adapt to changing network conditions and consistently control the queueing latency to the reference level of 20ms.

flows' throughputs are 1.86Mbps, 2.15Mbps, 1.80Mbps, 1.92Mbps and 2.09Mbps respectively, close to their fair share.

2) Heavy TCP traffic: In this test scenario, we increase the number of TCP flows to 50. With higher traffic intensity, the link utilization reaches 100% as clearly shown in Figure 5(b). The queueing delay, depicted in Figure 4(b), is controlled around the desired 20ms, unaffected by the increased traffic intensity. The effect of sawtooth fades in this heavy traffic case. The queueing delay fluctuates more evenly around the reference level. The average throughput reaches full capacity of 10Mbps as shown in Figure 5(b).

3) Mixture of TCP and UDP traffic: To demonstrate PIE's performance under persistent heavy congestion, we adopt a mixture of TCP and UDP traffic. More specifically, we have 5 TCP flows and 2 UDP flows (each sending at 6Mbps). The corresponding latency plot can be found in Figure 4(c). Again, PIE is able to contain the queueing delay around the reference level regardless the traffic mix while achieving 100% throughput shown in Figure 5(c).

4) Bursty Traffic: As the last functionality test, we show PIE's ability to tolerate bursts, whose maximum value is denoted by max_burst. We construct a test where one short lived UDP traffic sends at a peak rate of 25Mbps over a $burst\_len$ period of time. We set $burst\_len$ to be 100ms and 200ms respectively. We also set max_burst values to be 0 (i.e., no burst tolerance) and 100ms. The UDP flow starts sending at simulation time of 1s. Figure 6 plots the number of dropped packets as a function of the simulation time for four combinations of $burst\_len$ and max_burst. It is obvious from the graph that if the $burst\_len$ is less than max_burst, no packets are dropped. When $burst\_len$ equals to 200ms, the first 100ms of the burst are allowed into the queue and the PIE algorithm starts dropping only afterword. Similarly, if we set the PIE scheme to have no burst tolerance (i.e., max_burst = 0), the algorithm starts dropping as soon as the queue starts filling up.

Performance Evaluation and Comparison: The functions of PIE are verified above. This section evaluates PIE under dynamic traffic scenarios, compares its performance against RED and shows how PIE is better suited for controlling latency in today's Internet. The simulation topology is similar to the above. The core router runs either the RED or PIE scheme. The buffer size is 2MB for the two schemes. RED queue is configured with the following parameters: min_{th} = 20% of the queue limit, max_{th} = 80% of the queue limit, max_p = 0.1 and q_weight = 0.002. The PIE queue is configured with the same parameters as in the previous section.

5) Varying Link Capacity: This test starts with the core capacity of 100Mbps; 100 TCP flows start randomly in the initial second. At 50s, the available bandwidth drops to 20Mbps and jumps back to 100Mbps at 100s. The core routers queue limit is set to 2MB. Figure 7 plots the queuing latency experienced by the RED and PIE queues. The figure shows that both AQM schemes converge to equilibrium after a couple of seconds from the beginning of the simulation. The queues experience minimal congestion during the first 50s. RED operates around min_{th}, and the queue latency settles around 30ms accordingly. Similarly, PIE converges to the equilibrium value of 20ms. When available bandwidth drops to 20Mbps (at 50s), the queue size and latency shoot up for both RED and PIE queues. RED's queue size moves from min_{th} to max_{th} to accommodate higher congestion, and the queuing latency stays high around 300ms between 50s and 100s. On the other hand, PIE's drop probability quickly adapts, and in about three seconds, PIE is able to bring down the latency around the equilibrium value (20ms). At 100s, the available bandwidth jumps back to 100Mbps. Since congestion is reduced, RED's queue size comes down back to min_{th}, and the latency is reduced as well. PIE's drop probability scales down quickly, allowing PIE's latency to be back at the equilibrium value. Note that due to static configurations of RED's parameters, it cannot provide consistent performance under varying network conditions. PIE, on the other hand, automatically adapts itself and can provide steady latency control for varying network conditions.

6) Varying Traffic Intensity: We also verify both schemes' performance under varying network traffic load with the number of TCP flows ranging from 10 through 50. The simulation starts with 10 TCP flows and the number of TCP flows jumps to 30 and 50 at 50s and 100s, respectively. The traffic intensity reduces at 150s and 200s when number of flows drops back to 30 and 10, respectively. Figure 8 plots the queuing latency experienced under the two AQM schemes. Every time the traffic intensity changes, RED's operating queue size and latency are impacted. On the other hand, PIE quickly adjusts the dropping probability in a couple of seconds, and restore the control of the queuing latency to be around equilibrium value.

B. Testbed Experiments

We also implemented PIE in Linux Kernel. In this section, we evaluate PIE in the lab setup and compare it against CoDel whose design is the most up to date in Linux. The current implementation is on Kernel Version 3.5-rc1.
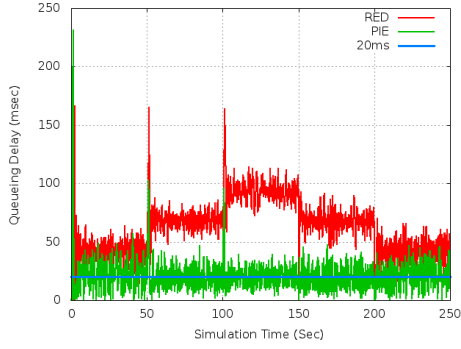
Fig. 8. PIE vs. RED Performance Comparison Under Varying Traffic Intensity: 0s-50s, traffic load is 10 TCP flows; 50s-100s, traffic load is 30 TCP flows;100s-150s, traffic load is increased to 50 TCP flows; traffic load is then reduced to 30 and 10 at 200s and 250s respectively. Due to static configured parameters, the queueing delay increases under RED as the traffic intensifies. The autotuning feature of PIE, however, allows the scheme to control the queueing latency quickly and effectively.
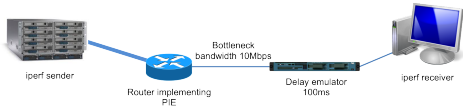

Fig. 9. The Testbed Setup: our experiment testbed consists of four unique Linux-based machines. The router implements AQM schemes.

Our experiment testbed consists of four unique Linux-based machines as shown in Figure 9. The sender is directly connected to the router with the PIE implementation. Hierarchical token bucket (htb) qdisc has been used to create a bandwidth constraint of 10Mbps. The router is connected to the receiver through another server. The delay is added in the forward direction using a delay emulator. Iperf tool is run on the sender and receiver to generate traffic. All measurements are done at the router. We obtain statistics and measure throughput at the router through the tc interface.

For all our lab test scenarios, we use the following PIE parameters PIE: $\tilde{\alpha}$ = 0.125, $\tilde{\beta}$ = 1.25, buffer size = 200 packets, T_{update} = 30ms. CoDel is used for comparison, whose parameters are set accordingly to the default: $interval$ = 100ms and $queue\_limit$ = 200 packets. Packet sizes are 1KB for both schemes.
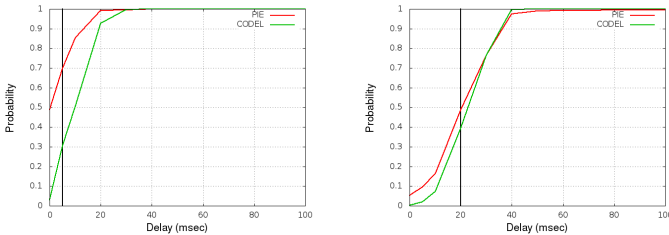
(a) Reference Delay = 5ms   (b) Reference Delay = 20ms

Fig. 10. Cdf of Queueing Delay Comparison Between PIE and CoDel: 20 TCP flows. When the reference delay is 5ms, 70% of the delays under PIE vs. 30% of delays under CoDel are less than 5ms. PIE and CoDel behave similarly when the reference delay is 20ms.

1) TCP Traffic: We evaluate the performance of PIE and CoDel in a moderately congested scenario. We use 20 NewReno TCP flows with RTT = 100ms and run the test for 100 seconds. Figure 10 plots the cdf curve of the queueing delay for PIE and CoDel when the
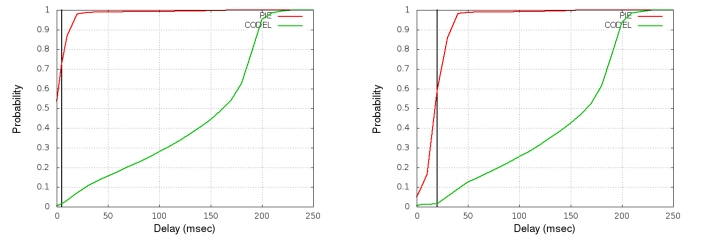
(a) Reference Delay = 5ms   (b) Reference Delay = 20ms

Fig. 11. Cdf of Queueing Delay Comparison Between PIE and CoDel: 5 TCP flows and 2 UDP flows. It is obvious that, under heavy congestion, CoDel cannot control the latency to the the target values while PIE behaves consistently according to the design.

reference delay is 5ms and 20ms respectively.[1] Both schemes are able to control the queueing delay reasonably well. When the target delay is 5ms, more than 90% packets under both schemes experience delays that are less than 20ms. It is clear that PIE performs better: 70% of the delays are less than 5ms while CoDel has only 30%. When the reference delay equals to 20ms, the performance of both schemes look similar. PIE still performs slightly better: 50% of packet delays are less than 20ms while only 40% of packet delays are less than 20ms under CoDel. For the 20ms target delay case, the throughput for PIE is 9.87Mbps vs. CoDel is 9.94Mbps. For the 5ms target delay case, the throughput for PIE is 9.66Mbps vs. CoDel is 9.84Mbps. CoDel's throughput is slightly better than PIE.

2) Mixture of TCP and UDP Traffic: In this test, we show the stability of both schemes in a heavily congested scenario. We setup 5 TCP flows and 2 UDP flows (each transmitting at 6Mbps). The 2 UDP flows result in a 20% oversubscription on the 10Mbps bottleneck. Figure 11 shows the cdf plot for the delay. Under the mixture of TCP and UDP traffic, it is obvious that CoDel cannot control the latency under the target values of 5ms and 20ms respectively. Majority of the packets experience long delays over 100ms. PIE, on the other hand, behaves consistently according to the design: with 70% less than the target of 5ms, and 60% less than the target of 20ms respectively. Vast majority of packets, close to 90%, do not experience delay that is more than twice of the target value. In this test, when the target delay equals to 20ms, the throughput for PIE is 9.88Mbps vs. CoDel is 9.91Mbps. When the target delay equals to 5ms, the throughput for PIE is 9.79Mbps vs. CoDel is 9.89Mbps. Throughputs are similar.

V. THEORETICAL ANALYSIS

We formulate our analysis model based on the TCP fluid-flow and stochastic differential equations originated in the work by Misra et al. [14] and [15]. We consider multiple TCP streams passing through a network that consists of a congested link with a capacity C_l. It is shown in [14] that the TCP window evolution can be approximated as

$$\frac{dW(t)}{dt} = \frac{1}{R(t)} - \frac{W(t)W(t-R(t))}{2\ R(t-R(t))}p(t-R(t)); \quad (1)$$

$$\frac{dq(t)}{dt} = \frac{W(t)}{R(t)}N(t) - C_l; \quad (2)$$

where $W(t)$ and $q(t)$ denote the TCP window size and the expected queue length at time t. The load factor, number of flows, is indicated

---

[1]There is a subtle difference in the definition of terms: in CoDel, the parameter, $target$, represents the target latency, while del_ref refers the reference equilibrium latency in PIE. For easy comparison, we would use "Reference Delay" to refer the target latency, $target$, in CoDel and the reference equilibrium latency, del_ref, in PIE.
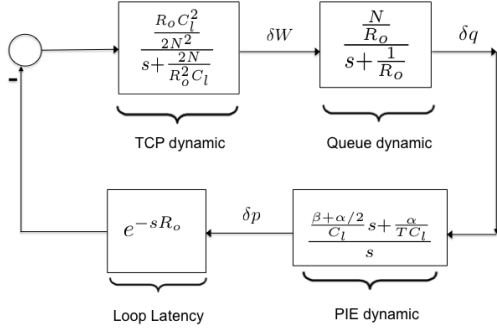
Fig. 12. The Feedback Loop of PIE: it captures TCP, Queue and PIE dynamics; and also models the RTT delay.
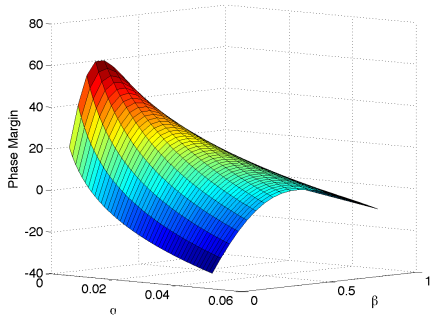

Fig. 13. Phase Margin as a Function of Parameters α and β: in order for a system to be stable, we need a phase margin above $0°$.

as $N(t)$. $R(t)$ represents the round trip time including the queueing delay. Note that $R(t)$ denotes the harmonic mean of the flows' round trip times. The drop or mark probability is indicated as $p(t)$.

AQM schemes determine the relationship between the drop or mark probability $p(t)$ and the queue length $q(t)$. This relationship in PIE is detailed in Section III-B. PIE increases its drop or mark probability based on current queueing delay and the delay moving trend. Using Bilinear Transformation, we can convert the PIE design into a fluid model as follows:

$$\tau(t) = \frac{q(t)}{C_l}; \tag{3}$$

$$\frac{dp(t)}{dt} = \alpha\frac{\tau(t) - \tau_{ref}}{T} + (\beta + \frac{\alpha}{2})\frac{d\tau(t)}{dt}; \tag{4}$$

where τ and τ_{ref} represent the queue latency and its equilibrium reference value, and T is the update interval, which equals to T_{update}. Note that although T does not show up directly in the discrete form of the algorithm, it does play a role in the overall system's behavior.

These fluid equations describe our control system's overall behavior from which we can derive the equilibrium and dynamic characteristics as follows.

A. Linearized Continuous System

When the system reaches steady state, the operating point (W_o, q_o, p_o) is defined by $\dot{W} = 0$, $\dot{q} = 0$ and $\tau = \tau_{ref}$ so that

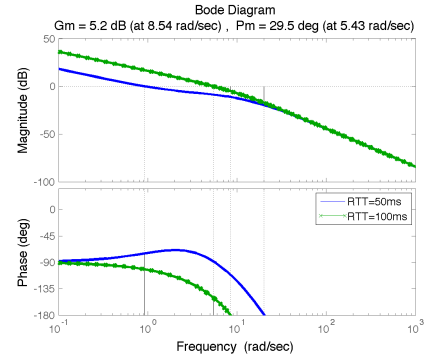$$W_o^2 p_o = 2 \text{ and } W_o = \frac{R_o C_l}{N}. \tag{5}$$

Fig. 14. An Illustration about Stability Margin for Different RTTs: for $R_o < R^+$, the gain of the system reduces and poles are moved towards high frequency. As a result, phase margin increases.

We can approximate our system's dynamics by their small-signal linearization about an operating point based on small perturbations: e.g. $W = W_o + \delta W$. Our system equations above lead to the following:

$$\delta\dot{q}(t) = \frac{N}{R_o} - \delta W(t) - \frac{1}{R_o}\delta q(t); \tag{6}$$

$$\delta\dot{W}(t) = -\frac{N}{R_o^2 C_l}(\delta W(t) - \delta W(t - R_o))$$
$$- \frac{R_o C_l^2}{2N^2}\delta p(t - R_o); \tag{7}$$

$$\delta\dot{p}(t) = \frac{\alpha}{TC_l} + \frac{\beta + \alpha/2}{C_l}\delta q(t). \tag{8}$$

From Equations (5) to (8), we can obtain transfer function in Laplace domain shown in Figure gain as:

$$\mathcal{G}(s) \approx -\frac{\frac{C_l}{2N}((\beta + \alpha/2)s + \frac{\alpha}{T})}{(s + \frac{2N}{R_o^2 C_l})(s + \frac{1}{R_o})}\frac{e^{-sR_o}}{s}$$

$$i.e. \approx -\frac{\kappa(j\omega/z_1 + 1)}{(j\omega/s_1 + 1)(j\omega/s_2 + 1)}\frac{e^{-j\omega R_o}}{j\omega}, \tag{9}$$

where $\kappa = \alpha R_o/(p_o T)$, $z_1 = \alpha/((\beta + \alpha/2)T)$, $s_1 = \sqrt{2p_o}/R_o$ and $s2 = 1/R_o$. Note that the loop scales with C/N, which can be derived from the drop probability p_o and R_o.

B. Determining Control Parameters α, β

Although many combinations of T, α and β would lead to system stability, we choose our parameter settings according to the following guideline. We choose T so that we sample the queueing latency two or three times per round trip time. Then, we set the values of α and β so that the Bode diagram analysis would yield enough phase margin. Figure 13 shows what phase margin would be for various values of α and β given that $T = 30$ms, $R^+ = 100$ms and $p^- = 0.01\%$. It is obvious that, in order to have system stability, we need to choose α and β values so that we have phase margin above $0°$.

Once we choose the values of α and β so that the feedback system in Equation (9) is stable for R^+ and p^-. For all systems with $R_o < R^+$ and $p_o > p^-$, the gain of the loop, $|\mathcal{G}(s)| = \alpha R_o/(p_o T), < \alpha R^+/(p^- T)$, $s_1 > \sqrt{2p^-}/R^+$ and $s_2 > 1/R^+$. Hence, if we make the system stable for R^+ and p^-, the system will be stable for all $R_o < R^+$ and $p_o > p^-$. Figure 14 illustrates this point showing how the phase margin improves from $29.5°$ to $105.0°$ when R_o is 50ms instead of R^+ of 100ms.

We could fix the values of α and β and still guarantee stability across various network conditions. However, we need to choose
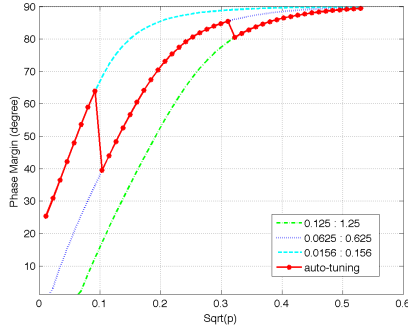
Fig. 15. Phase Margin as a Function of $\sqrt{p_o}$: three pairs of $\alpha : \beta$ settings. Lower values of α and β gives higher phase margin. Autotuning picks different pair for different p_o range to optimally tradeoff stability vs. response time.
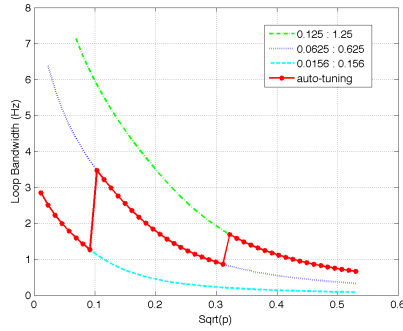

Fig. 16. Loop Frequency as a Function of $\sqrt{p_o}$: three pairs of $\alpha : \beta$ settings. Lower values of α and β has slower response time. Autotuning picks different pair for different p_o range to optimally tradeoff stability vs. response time.

conservative values of α and β to achieve stability. For example, we need to set $\alpha = 0.0156$ and $\beta = 0.156$ in the above case to guarantee a phase margin of $25°$ for $p_o \approx 0.01\%$. However, when the drop probability increases, the system response time would take a hit. Figure 15 shows the phase margin as a function of $\sqrt{p_o}$ [2] for $\alpha : \beta$ values of 0.125:1.25, (0.125/2):(1.25/2) = 0.0625:0.625, (0.125/8):(1.25/8) = 0.0156:0.156, respectively. Their corresponding loop bandwidths, which directly determine their response time, are shown in Figure 16. As shown in Figure 15, if we choose $\alpha : \beta$ values to be 0.0625:0.625 and 0.125:1.25, we don't have enough phase margin to ensure stability when $p_o < 1\%$, i.e. $\sqrt{p_o} < 0.1$. On the other hand, these two higher value pairs would lead to faster response time as depicted in Figure 16.

Auto-tuning in PIE tries to solve the above problem by adapting its control parameters α and β based on congestion levels. How congested a link is can be easily inferred from the drop probability p_o. When the network is lightly congested, say under 1%, we choose numbers that can guarantee stability. When the network is moderately congested, say under 10%, we can increase their values to increase system response time. When the network is heavily congested, we can increase their values even further without sacrificing stability. While the adjustment can be continuous, we choose discrete numbers for simplicity. As demonstrated in Figure 15 and 16, the auto-tuning design in PIE can improve the response time of the loop greatly without losing stability. Our tests results in Section IV als shows that

[2]We choose $\sqrt{p_o}$ instead of p_o because s_1 scales with $\sqrt{p_o}$.

auto-tuning works well in varying congestion environment.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have described PIE, a latency-based design for controlling bufferbloat in the Internet. The PIE design based its random dropping decisions not only on current queueing delay but also on the delay moving trend. In addition, the scheme self-tunes its parameters to optimize system performance. As a result, PIE is effective across diverse range of network scenarios. Our simulation studies, theoretical analysis and testbed results show that PIE can ensure low latency under various congestion situations. It achieves high link utilization while maintaining stability consistently. It is a light-weight, enqueing based design that works with both TCP and UDP traffic. The PIE design only requires low speed drop probability update, so it incurs very small overhead and is simple enough to implement in both hardware and software.

Going forward, we will study how to automatically set latency references based on link speeds: set low latency references for high speed links while being conservative for lower speed links. We will also explore efficient methods to provide weighted fairness under PIE. There are two ways to achieve this: either via differential dropping for flows sharing a same queue or through class-based fair queueing structure where flows are queued into different queues. There are pros and cons with either approach. We will study the tradeoffs between these two methods.

REFERENCES

[1] B. Turner, "Has AT&T Wireless Data Congestion Been Self-Inflicted?" [Online]. Available: BroughTurnerBlog
[2] J. Gettys, "Bufferbloat: Dark buffers in the internet," *IEEE Internet Computing*, vol. 15, pp. 95–96, 2011.
[3] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, Aug. 1993.
[4] W. Feng, K. Shin, D. Kandlur, and D. Saha, "The blue active queue management algorithms," *IEEE/ACM Transactions on Networking*, vol. 10, no. 4, pp. 513–528, Aug. 2002.
[5] C. V. Hollot, V. Misra, D. Towsley, and W. bo Gong, "On designing improved controllers for aqm routers support," in *Proceedings of IEEE Infocom*, 2001, pp. 1726–1734.
[6] S. Kunniyur and R. Srikant, "Analysis and design of an adaptive virtual queue (avq) algorithm for active queue management," in *Proceedings of ACM SIGCOMM*, 2001, pp. 123–134.
[7] B. Braden, D. Clark, J. Crowcroft, and et. al., "Recommendations on Queue Management and Congestion Avoidance in the Internet," RFC 2309 (Proposed Standard), 1998.
[8] K. Nichols and V. Jacobson, "A Modern AQM is just one piece of the solution to bufferbloat." [Online]. Available: http://queue.acm.org/detail.cfm?id=2209336
[9] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulaton of a fair queueing algorihtm," *Journal of Internetworking Research and Experience*, pp. 3–26, Oct. 1990.
[10] R. Pan, B. Prabhakar, F. Bonomi, and R. Olsen, "Approximate fair bandwidth allocation: a method for simple and flexible traffic management," in *Proceedings of 46th Annual Allerton Conference on Communication, Control and Computing*, 2008.
[11] G. Franklin, J. D. Powell, and A. Emami-Naeini, in *Feedback Control of Dynamic Systems*, 1995.
[12] R. Pan, B. Prabhakar, and et. al., "Data center bridging - congestion notification." [Online]. Available: http://www.ieee802.org/1/pages/802.1au.html
[13] "NS-2." [Online]. Available: http://www.isi.edu/nsnam/ns/
[14] V. Misra, W.-B. Gong, and D. Towsley, "Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red," in *Proceedings OF ACM SIGCOMM*, 2000, pp. 151–160.
[15] C. V. Hollot, V. Misra, D. Towsley, and W. bo Gong, "A control theoretic analysis of red," in *Proceedings of IEEE Infocom*, 2001, pp. 1510–1519.